Addressing Modes

The ‘Addressing Mode’ of an instruction is the means by which it gains access to the data which is operated on.

For some instructions, there may be two operands and a result and these could conceivably each be using different addressing modes.

The main addressing modes found on processors are:

Implied	no field is needed since this instruction always acts on the same register or location.
Register	a field specifies a register number
Absolute	a field is the memory address of the operand
PC relative	a field is an offset from the instruction at which the operand can be found
Immediate	a field is the actual operand (read only)
Register Indirect	a field specifies a register which contains the address of the operand.
Indexed	a field is an offset to be added to a specified register to generate the address of the operand.
Memory Indirect	a field is an absolute address of the first of a number of bytes in memory which hold the actual operand address.

The addressing modes of the following Intel instructions will be discussed in lectures:

```
  I, A            PC rel        R,A / A,R
-----------      --------      -------------
cmpl $0,den      jmp LAB5      cmpl %eax,den
movl $0,res      je LAB54      movl num,%eax
sarl $1,p
sall $1,den
```

In these Intel instructions, there is a maximum of two arguments, and where an operation, such as addition, requires two operands and a result, one of the second argument is both an operand and the result.

Example Assembly Language Program

Here is an example assembly language program using the Intel 32 bit mode available in the Intel 486 and Pentium processors. These have variable length instructions. This example is long division in binary of two positive integers.

The following psudo code in a fictional high level language performs the required long division algorithm, although there is one bug:

```
// Subroutine to do binary long division
int num;   // Numerator
int den;   // Denominator
int res;   // Result
int p;     // Working variable

divide()
{
  if (den == 0) goto error;

  p = 1;
  res = 0;

  while (den & 0x80000000)
    { den = den << 1;
      p = p << 1;
    }
  while (den != 0)
    {
      if (num >= den)
        { num = num - den;
          res = res | p;
        }
      den = den >> 1;
      p = p >> 1;
    }

   error: printf("Divide by zero\n");
}
```

The algorithm first tests the denominator to see if it is zero, and if so prints an error. Otherwise it enters the first loop which shifts the denominator left until its most significant bit is set. Then it enters the second loop, which tries to subtract the denominator and its submultiples, recording the successes in the result variable.

The code listings on the next two page are the input and the output of the assembler respectively.

Exercise: Find the bug.

Exercise: It is not really necessary to keep p in store when there are registers available. Optimise the code by keeping more of the variables in registers. The registers available are EAX, EBX, ECX and EDX.

Assembly language file

```
        .org 0x1000     # Gives address of region now being defined.
den:    .long 0         # Define storage for the variables.
num:    .long 0         # Each variable takes 4 bytes.
p:      .long 0
res:    .long 0

        .org 0x4000     # Give address of start of program code
LABC0:
        .string "Divide by zero\n"


.globl divide           # This makes the label ‘divide’ externally callable
divide:
        cmpl $0,den     # See if denominator is zero
        je LAB3         # yes, then branch to error routine.

        movl $1,p       # Set variable p to 1
        movl $0,res     # Set result to zero

LAB4:   movl den,%eax   # Test most significant bit of den
        andl $0x80000000,%eax
        testl %eax,%eax # Set Z and N flags
        je LAB5         # Branch if Z set out of the loop
        sall $1,den     # Shift arithmetic left the den
        sall $1,p       # Shift p left as well.
        jmp LAB4        # Branch back to top of first loop

LAB5:   cmpl $0,den     # Second loop, check den is zero
        je LAB8         # If so, brach to exit
        movl num,%eax
        cmpl %eax,den   # Compare num with den using EAX register
        jg LAB10        # If den is greater, skip next bit.

        movl den,%edx   # Load den to EDX register
        subl %edx,num   # Subtract EDX from the numerator
        movl p,%edx     # Load p to EDX register
        orl %edx,res    # Or EDX into res
LAB10:
        sarl $1,den     # Shift arithmetic right den (divide by two)
        sarl $1,p       # Ditto p
        jmp LAB5        # Jump to top of second loop

LAB8:                   # I should have branched over the error
        nop             # message instead of having a NOP here
LAB3:
        pushl $LABC0    # Push the address of the string onto the stack
        int $5          # Use system call number 5 to print the string
        addl $4,%esp    # Restore stack to old value by adding 4 to sp
LAB1:
        ret             # Return from the divide subroutine.

# END OF FILE
```

Output from the Assembler

```
0000 00000000                .org 0x1000     # Gives address of region now being defined.
     00000000
     00000000
     00000000
     00000000
1000 00000000        den:    .long 0         # Define storage for the variables.
1004 00000000        num:    .long 0         # Each variable takes 4 bytes.
1008 00000000        p:      .long 0
100c 00000000        res:    .long 0
1010 00000000                .org 0x4000     # Give address of start of program code
     00000000
     00000000
     00000000
     00000000
                     LABC0:
4000 44697669                .string "Divide by zero\n"
     64652062
     79207A65
     726F0A00
                     .globl divide           # This makes the label ‘divide’ external
                     divide:
4010 833D0010                cmpl $0,den     # See if denominator is zero
     000000
4017 7471                    je LAB3         # yes, then branch to error routine.
4019 C7050810                movl $1,p       # Set variable p to 1
     00000100
     0000
4023 C7050C10                movl $0,res     # Set result to zero
     00000000
     0000
402d A1001000        LAB4:   movl den,%eax   # Test most significant bit of den
     00
4032 25000000                andl $0x80000000,%eax
     80
4037 85C0                    testl %eax,%eax # Set Z and N flags
4039 7410                    je LAB5         # Branch if Z set out of the loop
403b C1250010                sall $1,den     # Shift arithmetic left the den
     000001
4042 C1250810                sall $1,p       # Shift p left as well.
     000001
4049 EBE2                    jmp LAB4        # Branch back to top of first loop
404b 833D0010        LAB5:   cmpl $0,den     # Second loop, check den is zero
     000000
4052 7435                    je LAB8         # If so, brach to exit
4054 A1041000                movl num,%eax
     00
4059 39050010                cmpl %eax,den   # Compare num with den using EAX register
     0000
405f 7F18                    jg LAB10        # If den is greater, skip next bit.
4061 8B150010                movl den,%edx   # Load den to EDX register
     0000
4067 29150410                subl %edx,num   # Subtract EDX from the numerator
     0000
406d 8B150810                movl p,%edx     # Load p to EDX register
     0000
4073 09150C10                orl %edx,res    # Or EDX into res
     0000
                     LAB10:
4079 C13D0010                sarl $1,den     # Shift arithmetic right den (divide by two)
     000001
4080 C13D0810                sarl $1,p       # Ditto p
     000001
4087 EBC2                    jmp LAB5        # Jump to top of second loop
                     LAB8:                   # I should have branched over the error
4089 90                      nop             # message instead of having a NOP here
                     LAB3:
408a 68004000                pushl $LABC0    # Push the address of the string onto the stack
     00
408f CD05                    int $5          # Use system call number 5 to print the string
4091 83C404                  addl $4,%esp    # Restore stack to old value by adding 4 to sp
                     LAB1:
4094 C3                      ret             # Return from the divide subroutine.
                     # END OF FILE
```

Direct Memory Access

Direct memory access is where the processor temporarily gives up control of the bus (i.e. being bus master) and allows the peripheral to take control. The peripheral generates addresses and control signals to transfer data to or from the memory directly. DMA is normally faster for bulk data transfer than programmed IO since every bus cycle carries useful data. The opposite, ‘programmed IO’ is slower because it requires each data word to be read from device/memory and then written to memory/device via a processor register with instruction fetch cycles in between.

Some IO devices contain their own DMA controller which is able to choose which buffer to access next according to events on the physical device. Other IO devices use a system provided DMA controller which simply shovels data.

The DMA controller requires the addresses of the from and to regions of memory and the number of bytes to transfer.

Typically, one of the source or destination will be an IO device data register.

Example of use - move data to an IO device.

```
LOAD  R1, =MyBufferStart
STORE R1, DMA_CONTROL_FROM_ADDRESS_REGISTER

LOAD  R1, =(MyBufferEnd-MyBufferStart+1)
STORE R1, DMA_CONTROL_LENGTH_REGISTER

LOAD  R1, =IO_DEVICE_DATA_REGISTER
STORE R1, DMA_CONTROL_TO_ADDRESS_REGISTER

LOAD  R1, =0x47
STORE R1, DMA_CONTROL_CONTROL_REGISTER
```

On this controller we take it that 0x47 is the value required to make it start transfering data with the source address being increased and destination address being held constant.

When finished, an interrupt is typically generated, but the processor can also poll the DMA_CONTROL_STATUS_REGISTER

Exercise: For what length blocks will using this DMA controller be more efficient than programmed IO ?

The Fetch Execute cycle.

1 Processor Reset.

- Power on reset
- Reset by switch

2 Check for external interrupts.

- If present, save current PC
- Load new fixed starting PC

3 Fetch an instruction.

- One byte or multiword instructions

4 Obey the instruction (using the ALU if needed).

5 If the instruction was a jump goto 2

6 Otherwise increment the to point to the next instruction and goto 2.

Interrupts

Interrupts stop the flow of the current program and cause the processor to jump to a new location which can handle the interrupt.

Interrupt action

1. Save the program counter and flags
2. Clear the I flag and set the P flag
3. Load the new PC

How do we return from them ?

1. Load PC from where saved
2. Restore flags
3. Continue processing .....

We can either continue
- With a restart of the last instruction
- With the next instruction

Interrupt Classes

There are three main classes of interrupt:
1 A device external to the processor.
2 Exception during processing an instruction.
3 The software interrupt, or service call.

Example Interrupt Sources

Class	Source or cause
1	UART Serial Port interrupt
1	Real-time clock interrupt
1	Disk drive interrupt
1	etc
2	Illegal instruction (bad opcode)
2	Illegal address of operand
2	Emulation of missing coprocessor
2	VM translation fault (page paged out)
2	Privilege violation
3	Trap (system call)

ASCII Character Set

```
----------------------------
      2   3   4   5   6   7
----------------------------
 0  | sp  0   @   P   ‘   p
 1  | !   1   A   Q   a   q
 2  | "   2   B   R   b   r  The ASCII code for a particular
 3  | #   3   C   S   c   s  character is 16 times the number at
 4  | $   4   D   T   d   t  the top plus the number down
 5  | %   5   E   U   e   u  the side.
 6  | &   6   F   V   f   v
 7  | ’   7   G   W   g   w
 8  | (   8   H   X   h   x
 9  | )   9   I   Y   i   y
 A  | *   :   J   Z   j   z
 B  | +   ;   K   [   k   {
 C  | ,   <   L   \   l   |
 D  | -   =   M   ]   m   }
 E  | .   >   N   ^   n   ~
 F  | /   ?   O   _   o  del
----------------------------
```

Text may be represented using the American Standard Code for Information Interchange (ASCII) coding. Seven bits are used, giving 128 possible codes. Codes from 0 to 31 are non-printing control characters, such as

```
      warning bell = 7
         backspace = 8
   carriage return = 13
          linefeed = 10
```

Keyboards generate keyscan codes instead of direct ASCII characters. So that shift, control and alternate keys can be supported, each key produces a code when it is pressed and another when it is released.

Exercise:

1. Give the sequence of keyscan codes generated by a keyboard and the resulting ASCII bytes that would occur when the words

   Banana Nana <CR>

are typed, leaving the cursor at the start of the next line. (Make up your own key scan codes).

Representing Data Structures

```
datatype rec = node of int * int * rec
         |     leaf of int;

val example = node(4, 5, node(6, 7, leaf(8)));
```

Address	Value	Comment
0xF30	2	Constructor tag for a leaf
0xF34	8	Integer 8
0xF3C	1	Constructor tag for a node
0xF40	6	Integer 6
0xF44	7	Integer 7
0xF48	0xF30	Address of inner node
0x1000	1	Constructor tag for a node
0x1004	4	Integer 4
0x1008	5	Integer 5
0x100C	0xF3C	Address of inner node

Representing Floating Point Numbers

Floating point numbers are represented using three fields, a mantissa, a sign flag and an exponent, all typically packed into the 32 bit word. The exponent is generally to the base two. The exponent can be in two's complement to allow numbers smaller than +/- 1/2.

For example, a 32 bit word may use the most significant bit as the sign, the next seven bits as the exponent (-64 to +63) and the next 24 bits as the mantissa with the first mantissa bit having significance 0.5.

Some examples

```
  ----------------------------------------------------------------
   Decimal                                                 Hex
   Value    S Exponent          Mantissa           representation
  ----------------------------------------------------------------
     0      0 000 0000 0000 0000 0000 0000 0000 0000   0x0000_0000
     1      0 000 0001 1000 0000 0000 0000 0000 0000   0x0180_0000
    -1      1 000 0001 1000 0000 0000 0000 0000 0000   0x8180_0000
     2      0 000 0010 1000 0000 0000 0000 0000 0000   0x0280_0000
     2.5    0 000 0010 1010 0000 0000 0000 0000 0000   0x02A0_0000
     0.25   0 111 1111 1000 0000 0000 0000 0000 0000   0x7F80_0000
  ----------------------------------------------------------------
```

Note that bit 23 is set for all numbers except zero and numbers very close to zero. This is the 'normal form' for them and generally offers greatest precision, although a few numbers may be accurately represented not in normal form. IEEE standard floating point is slightly more complex since it does not store bit 23, but implies it unless the exponent is at its most negative value.

A 24 bit mantissa gives 24 x log10(2) = 7 decimal digits of precision.

The maximum value is approximately 2 ** 63 which is 9.2E18.

Interrupt Driven I/O

A typical output device contains a small amount of output buffering and this is augmented using the main store of the host computer. Data generated by the application on the host is stored in a circular buffer in the host computer and taken out of the circular buffer under interrupt control.

The circular buffer is defined by two addresses, TX_Buffer_Start and TX_Buffer_End and is accessed via two pointers TX_In_Pointer and TX_Out_Pointer. When these contain the same value, the buffer is deemed empty.

The output device will generate an interrupt when it is ready for data and when its interrupt enable control bit is set. We will assume that the transmit interrupt enable flag is in Bit 1 of the device's control register.

Here is the code to place a byte from R0 in the output buffer:

```
OutputByte:        LOAD  R1, TX_In_Pointer
                   STORE R0, (R1)
                   CMP   R1, =TX_Buffer_End
                   BNE   L1
                   LOAD  R1, =TX_Buffer_Start
                   BRA   L2
        L1:        ADD   R1, =1
        L2:        STORE R1, TX_In_Pointer
                   LOAD  R1, Control_Register
                   OR    R1, =2
                   STORE R1, Control_Register
                   RTS
```

Here is interrupt service routine that empties the buffer

```
ISR:               LOAD  R1, TX_Out_Pointer
                   LOAD  R2, TX_In_Pointer
                   CMP   R1, R2
                   BNE   SomethingToSend
                   LOAD  R1, Control_Register
                   AND   R1, =0xFD
                   STORE R1, Control_Register
                   RTS

SomethingToSend:   LOAD  R0, (R1)
                   STORE R0, Data_Register
                   CMP   R1, =TX_Buffer_End
                   BNE   LA1
                   LOAD  R1, =TX_Buffer_Start
                   BRA   LA2
        LA1:       ADD   R1, =1
        LA2:       STORE R1, TX_In_Pointer
                   RTS
```

Interrupt Driven I/O continued

Note that the equals sign has been used to specify immediate addressing in this assembly language, whereas in the Intel language, the dollars sign is used. In these exercises, do now worry about the exact syntax of the assembler you write. If you wish, draw flowcharts instead.

Exercise: Modify the transmit routine called OutputByte so that it waits for the circular buffer to be non-full before proceeding. This will cause the output routine to degenerate to the performance of a polled system if data is generated for an extended period at a rate faster than can be consumed.

Exercise: Given that interrupts for reception are enabled when bit 3 of the control register is set to a one and there is data ready, sketch the equivalent code fragments (InputByte and ISR) required to perform input from this IO device using interrupts. This is easier since interrupts do not have to be switched on and off in the device. Assume that there is another circular buffer available and this is defined by two addresses, RX_Buffer_Start and RX_Buffer_End and is accessed via two pointers RX_In_Pointer and RX_Out_Pointer. When these contain the same value, this buffer is also deemed empty.

What happens if your input buffer overflows ?

Exercise: Given that the status register bits described in the polled I/O handout are also present in the status register, but that the interrupt service routine does not know whether it is a transmit or receive event that calls it, write a single ISR that works with both OutputByte and InputByte.

Summary: Types of Computer Memory

Store within the Processor:

RS Latch. The most basic electrical storage cell, made of two cross coupled inverters or inverting gates.

D-type or JK Flip-flop. A one bit store with an edge-triggered input.

Register. A number of flip-flops arranged side-by-side to hold a multi-bit word.

Register File. A number of registers arranged with multiplexors so that a set of register select input bits can select one of the registers for reading or writing. A dual port register file has two sets of select inputs and allows access to two registers at once.

Main and Cache Store Elements:

Static RAM (SRAM), has one RS latch per bit.

Dynamic RAM (DRAM) has one capacitor per bit, and needs refresh operations. Normally also has multiplexed address bus.

Read Only Memory (ROM) contents do not change when power is removed and so can be used for booting. Standard ROM is mask programmed at manufacture.

Eraseable field programmable ROM (EPROM) uses static charge storage on well-insulated transistor gates to hold info. Erased by exposure to UV light through window in top of package.

Electrically erasable ROM (EEROM) again uses static charge storage on floating gates, but the insulation is exceptionally thin to allow cold electron tunneling on and off of the gate when higher than normal voltages are applied.

Flash Memory is EEROM of a higher density (one transistor per bit) but where erasure (or clearing) of bits can only be done in small blocks.

Backing Store Elements:

Hard Disk

Floppy Disk

Tapes

- all three use magnetisation of a thin iron oxide layer.

Small System Memory Map Decoder

The following text is the HDL design for a chip which decodes
the memory map for a simple 8 bit microprocessor.

```
// These input definitions bring signals onto the decoder chip
pin 1 = a15;
pin 2 = a14;
pin 3 = a13;

// Output Definitions - Active low to enable the addressed device
pin 19 = rom_enable_bar;
pin 18 = ram_enable_bar;
pin 17 = uart_enable_bar;



// Equations for the following memory map
//      RAM at 0x0000 to 0x3FFF
//    EPROM at 0xC000 to 0xFFFF
//     UART at 0xA000 to 0xA007

  assign rom_enable_bar = !( a15 &  a14);
  assign ram_enable_bar = !(!a15 & !a14);
  assign uart_enable_bar = !(a15 & !a14 & a13);
// End of device
```

In the above:

```
         & denotes an AND gate
         ! denotes an INVERTER
  and    # would denote an OR gate
  and    ^ would denote an XOR gate, if one was used.
```

There are no wait states here. All devices go as fast as the processor can.

The equivalent circuit diagram will be given in lectures and can be drawn below.

Small System Decoder Exercises:

1. Given that the UART device decodes address lines A0 to A2 itself, as shown on the main circuit diagram, for access to its 8 internal registers, how many copies of the UART registers are present overall in the memory map.

2. It is desired that the RAM memory is addressed at zero, yet the processor resets also so that its PC is zero and so would instantly crash when first powered up (since the RAM will have random contents). To overcome this problem, an RS latch is added so that, on reset, when the latch is clear, the EPROM may be accessed at zero (and all other addresses), but that the processor can set this RS latch using its first write cycle to cause the memory map to have its normal mapping once the processor has jumped to its correct location. Give the circuit diagram and example instructions for the EPROM. (This exercise will be worked in lectures).

3. (Longer) For the basic situation in 2 above, redesign the system so that the processor copies the contents of ROM to RAM at address zero upwards, then disables the ROM completely (with a magic write to a certain address) and then continues executing normally out of RAM. N.B. An approach such as this is often used when the ROM access time is much greater than the RAM.

In order to answer 2 and 3, it will be necessary to bring the system reset wire to the decoder logic. To build the flip-flop, an RS latch made of two NOR gates can be used. Answers may be given as circuits or in any HDL. In HDL, the following code makes an RS latch:

```
assign q    = !(reset | qbar);
assign qbar = !(set  | or q);
```

Microsoft MSDOS Disk Format

The first 13 logical sectors of the disk are used as 1 to 3:

- [1.] The boot block. 1 sector.
- [2a.]The file allocation table (FAT) 3 sectors.
- [2b.] A copy of the FAT 3 sectors.
- [3.] The root directory 5 sectors.
- [4.] The dynamically allocated remainder

Two copies of the FAT are present for resilience.

Boot block gives details to establish logical to the physical mapping for the rest of the drive, where typically sectors are grouped in pairs to form blocks, which are then allocated to files or directories. With a sector size of 512 bytes, the smallest file uses up one 32 byte directory slot and 1K bytes of data storage.

The file allocation table (FAT) chains (logical) sectors and also shows freespace. MSDOS directory slots are 32 bytes long and each hold a file or subdirectory pointer. One slot per disk is used as a volume label.

There are 512/32 = 16 slots per sectore and in each slot bytes are used as follows

0-7	File name
8-10	Filename extension
11	Attribute flags 1=Read only, 2=Hidden, 4=System file, 8=Volume label 16=subdirectory, 32=Archived flag.
12-21	Reserved
22-23	Time of modification
24-25	Date of modification
26-27	Starting logical sector number
28-31	File size in bytes

Here is an example directory listing.

```
   HTCORE    CV     8:57:52  18-july-1994  6005    (cluster=52)
   LINKCTL   CV     8:57:56  18-july-1994  6373    (cluster=58)
   TAB       C      14:52:48  18-july-1994  34078  (cluster=265)
   SYSDES    N      17:18:48  15-april-1995  2581  (cluster=371)
Displayed total 358987 bytes.  354304 bytes free.
```

```
48 54 43 4F 52 45 20 20 43 56 20 00 00 00 00 00   HTCORE  CV .....
00 00 00 00 00 00 3A 47 F2 1C 34 00 75 17 00 00   ......:G..4.u...
4C 49 4E 4B 43 54 4C 20 43 56 20 00 00 00 00 00   LINKCTL CV .....
00 00 00 00 00 00 3C 47 F2 1C 3A 00 E5 18 00 00   ......<G..:.....
54 41 42 20 20 20 20 20 43 20 20 20 00 00 00 00   TAB     C   ....
00 00 00 00 00 00 98 76 F2 1C 09 01 1E 85 00 00   .......v........
53 59 53 44 45 53 20 20 4E 20 20 20 00 00 00 00   SYSDES  N   ....
00 00 00 00 00 00 58 8A 8F 1E 73 01 15 0A 00 00   ......X...s.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ...............
```

CPU Flags and Two’s Complement Arithmetic

Processor Flags

```
N    Negative - the most significant bit of the result

Z    Zero - a one if al result bits (ignoring carry) are zero

C    Carry - set if there is a carry from the left most bit

V    Overflow - set if carry in and out of msb are different.
```

Addition - Unsigned - Without carry

```
    00101   5
  + 00111   7
 -------------
 0  01100   12   No carry, so answer correct
 -------------
```

Addition - Unsigned - With carry

```
    11110   30
  + 00111   7
 -------------   Carry indicates answer is 32 too small
  1 00101   5    37 cannot be held in 5 bits
 -------------
```

Subtraction - Unsigned - No Borrow

```
    11110   30
  + 11001   -7   Add two’s complement to effect subtraction
 -------------
  1 10111   23   Correct answer, carry of one indicates no borrowing
 -------------   needed.
```

Subtraction - Unsigned - With Borrow

```
    00111   7
  + 00010  -30   Add two’s complement to effect subtraction
 -------------
 0  01001  9    Carry zero implies a borrow of 32 was needed.
 -------------
```

Second sheet - Signed Two’s Complement

Addition - Signed - No overflow

```
      00101   5
    + 00111   7
   -------------
   0  01100   12   Carry into and out of msb the same so no overflow
   -------------
      0
```

Addition - Signed - With overflow

```
      01010   10
    + 00111   7
   -------------    17 cannot be held in 5 bit two’s complement
   0  10001   -15   Carry into and out of msb different.
   -------------
      1
```

Subtraction - Signed - No overflow

```
      01010   10
    + 11001   -7
   -------------
   1  00011   3   Correct answer, carries both one, no overflow.
   -------------
      1
```

Subtraction - Signed - With overflow

```
      10110   -10  (-10)
    + 10110   -10  (Take away another 10)
   -------------
   1  01100    12  Wrong answer.
   -------------
      0
```

Exercise: Two numbers can be compared by subtracting them
and looking at the resulting settings of NZC and V. The rest
of the result may be ignored. Give the two Boolean functions of these
flags that can be used to see if one operand is less than the other,
one function for signed arithmetic and the other for unsigned.

Example of Polled IO

In polled input or output, the processor repeatedly reads
the status of a device to see if the device is ready for the next
data transfer.

In this example, the device has a status register in which bit 5
is non-zero if the device can accept data and bit 4 is non-zero
if there is a byte ready to read.

Here is the output routine

```
SendByte:  LOAD R1, StatusReg
           AND  R1, =32
           BEQ  SendByte
           STORE R0, DataReg
           RTS
```

Here is the input routine

```
GetByte:   LOAD R1, StatusReg
           AND  R1, =16
           BEQ  SendByte
           LOAD R0, DataReg
           RTS
```

Exercise: Sketch a graph showing the time wasted by polling an
I/O device as the relative speeds of the device and the processor
or program sourcing or sinking the data are changed. When is polling
useful ? Is polling used on multiuser, timesharing computers ?

Exercise: A new version of the device being controlled by this
software becomes available which includes 16 byte FIFO stores
between the processor bus interface and the physical I/O device.
How would these be structured and how does this affect your graph above ?

Exercise: When moody Jim Smith uses the hardware device that is
controlled by these polling routines, he uses it in a strange way such
that it alternates between two modes, one which generates large
numbers of bytes before it will consume any and one which consumes
all bytes that are available before making any new ones. What might
happens to naive control software in this case ? How could this be
overcome ?

Exercise: What happens to the processor’s data cache during polled
input or output ?